O'REILLY®

Second Edition

Data Science at the Command Line

Obtain, Scrub, Explore, and Model Data with Unix Power Tools

Jeroen Janssens

Foreword by Tim O'Reilly

Data Science at the Command Line
by Jeroen Janssens

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Sarah Grey
Production Editor: Kate Galloway
Copyeditor: Arthur Johnson
Proofreader: Shannon Turlington
Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2014:	First Edition
August 2021:	Second Edition

Revision History for the Second Edition

2021-08-17: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492087915* for release details.

978-1-492-08791-5

[LSI]

Table of Contents

Foreword

It was love at first sight.

It must have been around 1981 or 1982 that I got my first taste of Unix. Its command-line shell, which uses the same language for single commands and complex programs, changed my world, and I never looked back.

I was a writer who had discovered the joys of computing, and regular expressions were my gateway drug. I'd first tried them in the text editor in HP's RTE operating system, but it was only when I came to Unix and its philosophy of small cooperating tools with the command-line shell as the glue that tied them together that I fully understood their power. Regular expressions in `ed`, `ex`, `vi` (now `vim`), and `emacs` were powerful, sure, but it wasn't until I saw how `ex` scripts unbound became `sed`, the Unix stream editor, and then AWK, which allowed you to bind programmed actions to regular expressions, and how shell scripts let you build pipelines not only out of the existing tools but out of new ones you'd written yourself, that I really got it. Programming is how you speak with computers, how you tell them what you want them to do, not just once, but in ways that persist, in ways that can be varied like human language, with repeatable structure but different verbs and objects.

As a beginner, other forms of programming seemed more like recipes to be followed exactly—careful incantations where you had to get everything right—or like waiting for a teacher to grade an essay you'd written. With shell programming, there was no compilation and waiting. It was more like a conversation with a friend. When the friend didn't understand, you could easily try again. What's more, if you had something simple to say, you could just say it with one word. And there were already words for a whole lot of the things you might want to say. But if there weren't, you could easily make up new words. And you could string together the words you learned and the words you made up into gradually more complex sentences, paragraphs, and eventually get to persuasive essays.

Almost every other programming language is more powerful than the shell and its associated tools, but for me at least, none provides an easier pathway into the programming mindset, and none provides a better environment for a kind of everyday conversation with the machines that we ask to help us with our work. As Brian Kernighan, one of the creators of AWK as well as the coauthor of the marvelous book *The Unix Programming Environment*, said in an interview with Lex Fridman, "[Unix] was meant to be an environment where it was really easy to write programs." [00:23:10] Kernighan went on to explain why he often still uses AWK rather than writing a Python program when he's exploring data: "It doesn't scale to big programs, but it does pretty darn well on these little things where you just want to see all the somethings in something." [00:37:01]

In *Data Science at the Command Line*, Jeroen Janssens demonstrates just how powerful the Unix/Linux approach to the command line is even today. If Jeroen hadn't already done so, I'd write an essay here about just why the command line is such a sweet and powerful match with the kinds of tasks so often encountered in data science. But he already starts out this book by explaining that. So I'll just say this: the more you use the command line, the more often you will find yourself coming back to it as the easiest way to do much of your work. And whether you're a shell newbie, or just someone who hasn't thought much about what a great fit shell programming is for data science, this is a book you will come to treasure. Jeroen is a great teacher, and the material he covers is priceless.

— *Tim O'Reilly*
May 2021

Preface

Data science is an exciting field to work in. It's also still relatively young. Unfortunately, many people, and many companies as well, believe that you need new technology to tackle the problems posed by data science. However, as this book demonstrates, many things can be accomplished by using the command line instead, and sometimes in a much more efficient way.

During my PhD program, I gradually switched from using Microsoft Windows to using Linux. Because this transition was a bit scary at first, I started with having both operating systems installed next to each other (known as a dual-boot). The urge to switch back and forth between Microsoft Windows and Linux eventually faded, and at some point I was even tinkering around with Arch Linux, which allows you to build up your own custom Linux machine from scratch. All you're given is the command line, and it's up to you what to make of it. Out of necessity, I quickly became very comfortable using the command line. Eventually, as spare time got more precious, I settled down with a Linux distribution known as Ubuntu because of its ease of use and large community. However, the command line is still where I'm spending most of my time.

It actually wasn't too long ago that I realized that the command line is not just for installing software, configuring systems, and searching files. I started learning about tools such as `cut`, `sort`, and `sed`. These are examples of command-line tools that take data as input, do something to it, and print the result. Ubuntu comes with quite a few of them. Once I understood the potential of combining these small tools, I was hooked.

After earning my PhD, when I became a data scientist, I wanted to use this approach to do data science as much as possible. Thanks to a couple of new, open source command-line tools including `xml2json`, `jq`, and `json2csv`, I was even able to use the command line for tasks such as scraping websites and processing lots of JSON data.

In September 2013, I decided to write a blog post titled "7 Command-Line Tools for Data Science". To my surprise, the blog post got quite some attention, and I received a lot of suggestions of other command-line tools. I started wondering whether the blog post could be turned into a book. I was pleased that, some 10 months later, and with the help of many talented people (see the acknowledgments), the answer was yes.

I am sharing this personal story not so much because I think you should know how this book came about, but because I want to you know that I had to learn about the command line as well. Because the command line is so different from using a graphical user interface, it can seem scary at first. But if I could learn it, then you can as well. No matter what your current operating system is and no matter how you currently work with data, after reading this book you will be able to do data science at the command line. If you're already familiar with the command line, or even if you're already dreaming in shell scripts, chances are that you'll still discover a few interesting tricks or command-line tools to use for your next data science project.

What to Expect from This Book

In this book, we're going to obtain, scrub, explore, and model data—a lot of it. This book is not so much about how to become *better* at those data science tasks. There are already great resources available that discuss, for example, when to apply which statistical test or how data can best be visualized. Instead, this practical book aims to make you more *efficient* and *productive* by teaching you how to perform those data science tasks at the command line.

While this book discusses more than 90 command-line tools, it's not the tools themselves that matter most. Some command-line tools have been around for a very long time, while others will be replaced by better ones. New command-line tools are being created even as you're reading this. Over the years, I have discovered many amazing command-line tools. Unfortunately, some of them were discovered too late to be included in the book. In short, command-line tools come and go. But that's OK.

What matters most is the underlying idea of working with tools, pipes, and data. Most command-line tools do one thing and do it well. This is part of the Unix philosophy, which makes several appearances throughout the book. Once you have become familiar with the command line, know how to combine command-line tools, and can even create new ones, you have developed an invaluable skill.

Changes for the Second Edition

While the command line as a technology and as a way of working is timeless, some of the tools discussed in the first edition have either been superseded by newer tools (e.g., `csvkit` has largely been replaced by `xsv`) or abandoned by their developers (e.g., `drake`), or they've been suboptimal choices (e.g., `weka`). I have learned a lot since the first edition was published in October 2014, either through my own experience or as a result of the useful feedback from my readers. Even though the book is quite niche because it lies at the intersection of two subjects, there remains a steady interest from the data science community, as evidenced by the many positive messages I receive almost every day. By updating the first edition, I hope to keep the book relevant for at least another five years. Here's a nonexhaustive list of changes I have made:

- I replaced `csvkit` with `xsv` as much as possible. `xsv` is a faster alternative to working with CSV files.
- In Chapters 2 and 3, I replaced the VirtualBox image with a Docker image. Docker is a faster and more lightweight way of running an isolated environment.
- I now use `pup` instead of `scrape` to work with HTML. `scrape` is a Python tool I created myself. `pup` is much faster, has more features, and is easier to install.
- Chapter 6 has been rewritten from scratch. Instead of `drake`, I now use `make` to do project management. `drake` is no longer maintained, and `make` is much more mature and very popular with developers.
- I replaced `Rio` with `rush`. `Rio` is a clunky Bash script I created myself. `rush` is an R package that is a much more stable and flexible way of using R from the command line.
- In Chapter 9 I replaced Weka and BigML with Vowpal Wabbit (`vw`). Weka is old, and the way it is used from the command line is clunky. BigML is a commercial API that I no longer want to rely on. Vowpal Wabbit is a very mature machine learning tool that was developed at Yahoo! and is now at Microsoft.
- Chapter 10 is an entirely new chapter about integrating the command line into existing workflows, including Python, R, and Apache Spark. In the first edition I mentioned that the command line can easily be integrated with existing workflows but never delved into the topic. This chapter fixes that.

How to Read This Book

In general, I advise you to read this book in a linear fashion. Once a concept or command-line tool has been introduced, chances are that I employ it in a later chapter. For example, in Chapter 9, I make heavy use of `parallel`, which is discussed extensively in Chapter 8.

Data science is a broad field that intersects many other fields such as programming, data visualization, and machine learning. As a result, this book touches on many interesting topics that unfortunately cannot be discussed at great length. At the end of each chapter, I provide suggestions for further exploration. It's not required that you read this material in order to follow along with the book, but if you are interested, just know that there's much more to learn.

Who This Book Is For

This book makes just one assumption about you: that you work with data. It doesn't matter which programming language or statistical computing environment you're currently using. The book explains all the necessary concepts from the beginning.

It also doesn't matter whether your operating system is Microsoft Windows, macOS, or some flavor of Linux. The book comes with a Docker image, which is an easy-to-install virtual environment. It allows you to run the command-line tools and follow along with the code examples in the same environment as this book was written. You don't have to waste time figuring out how to install all the command-line tools and their dependencies.

The book contains some code in Bash, Python, and R, so it's helpful if you have some programming experience, but it's by no means required to follow along with the examples.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, directory names, and filenames.

`Constant width`
: Used for code and commands, as well as within paragraphs to refer to command-line tools and their options.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this at *https://oreil.ly/data-science-at-cl*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book. The author also maintains a version of the book online.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments for the Second Edition (2021)

Seven years have passed since the first edition came out. During this time, and especially during the last 13 months, many people have helped me. Without them, I would have never been able to write a second edition.

I was once again blessed with three wonderful editors at O'Reilly. I would like to thank Sarah "Embrace the deadline" Grey, Jess "Pedal to the metal" Haberman, and Kate "Let it go" Galloway. Their middle names say it all. With their incredible help, I was able to embrace the deadlines, put the pedal to metal when it mattered, and eventually let it go. I'd also like to thank their colleagues Angela Rufino, Arthur Johnson, Cassandra Furtado, David Futato, Helen Monroe, Karen Montgomery, Kate Dullea, Kristen Brown, Marie Beaugureau, Marsee Henon, Nick Adams, Regina Wilkinson, Shannon Cutt, Shannon Turlington, and Yasmina Greco, for making the collaboration with O'Reilly such a pleasure.

Despite having an automated process to execute the code and paste back the results (thanks to R Markdown and Docker), the number of mistakes I was able to make is impressive. Thank you Aaditya Maruthi, Brian Eoff, Caitlin Hudon, Julia Silge Mike Dewar, and Shane Reustle for reducing this number immensely. Of course, any mistakes left are my responsibility.

Marc Canaleta deserves a special thank you. In October 2014, shortly after the first edition came out, Marc invited me to give a one-day workshop about *Data Science at the Command Line* to his team at Social Point in Barcelona. Little did we both know that many workshops would follow. It eventually led me to start my own company: Data Science Workshops. Every time I teach, I learn something new. They probably don't know it, but each student has had an impact, in one way or another, on this book. To them I say: thank you. I hope I can teach for a very long time.

Captivating conversations, splendid suggestions, and passionate pull requests. I greatly appreciate each and every contribution by following generous people: Adam Johnson, Andre Manook, Andrea Borruso, Andres Lowrie, Andrew Berisha, Andrew Gallant, Andrew Sanchez, Anicet Ebou, Anthony Egerton, Ben Isenhart, Chris Wiggins, Chrys Wu, Dan Nguyen, Darryl Amatsetam, Dmitriy Rozhkov, Doug

Needham, Edgar Manukyan, Erik Swan, Felienne Hermans, George Kampolis, Giel van Lankveld, Greg Wilson, Hay Kranen, Ioannis Cherouvim, Jake Hofman, Jannes Muenchow, Jared Lander, Jay Roaf, Jeffrey Perkel, Jim Hester, Joachim Hagege, Joel Grus, John Cook, John Sandall, Joost Helberg, Joost van Dijk, Joyce Robbins, Julian Hatwell, Karlo Guidoni, Karthik Ram, Lissa Hyacinth, Longhow Lam, Lui Pillmann, Lukas Schmid, Luke Reding, Maarten van Gompel, Martin Braun, Max Schelker, Max Shron, Nathan Furnal, Noah Chase, Oscar Chic, Paige Bailey, Peter Saalbrink, Rich Pauloo, Richard Groot, Rico Huijbers, Rob Doherty, Robbert van Vlijmen, Russell Scudder, Sylvain Lapoix, TJ Lavelle, Tan Long, Thomas Stone, Tim O'Reilly, Vincent Warmerdam, and Yihui Xie.

Throughout this book, and especially in the footnotes and appendix, you'll find hundreds of names. These names belong to the authors of the many tools, books, and other resources on which this book stands. I'm incredibly grateful for their hard work, regardless of whether that work was done 50 years or 50 days ago.

Above all, I would like to thank my wife Esther, my daughter Florien, and my son Olivier for reminding me daily what truly matters. I promise it'll be a few years before I start writing the third edition.

Acknowledgments for the First Edition (2014)

First of all, I'd like to thank Mike Dewar and Mike Loukides for believing that my blog post, "7 Command-Line Tools for Data Science", which I wrote in September 2013, could be expanded into a book.

Special thanks to my technical reviewers Mike Dewar, Brian Eoff, and Shane Reustle for reading various drafts, meticulously testing all the commands, and providing invaluable feedback. Your efforts have improved the book greatly. Any remaining errors are entirely my own responsibility.

I had the privilege of working with three amazing editors: Ann Spencer, Julie Steele, and Marie Beaugureau. Thank you for your guidance and for being such great liaisons with the many talented people at O'Reilly. Those people include Laura Baldwin, Huguette Barriere, Sophia DeMartini, Yasmina Greco, Rachel James, Ben Lorica, Mike Loukides, and Christopher Pappas. There are many others whom I haven't met because they are operating behind the scenes. Together they ensured that working with O'Reilly has truly been a pleasure.

This book discusses more than 80 command-line tools. Needless to say, without these tools, this book wouldn't have existed in the first place. I'm therefore extremely grateful to all the authors who created and contributed to these tools. The complete list of authors is unfortunately too long to include here; they are mentioned in the Appendix. Thanks especially to Aaron Crow, Jehiah Czebotar, Christoph Groskopf,

Dima Kogan, Sergey Lisitsyn, Francisco J. Martin, and Ole Tange for providing help with their amazing command-line tools.

Eric Postma and Jaap van den Herik, who supervised me during my PhD program, deserve special thanks. Over the course of five years they taught me many lessons. Although writing a technical book is quite different from writing a PhD thesis, many of those lessons proved to be very helpful in the past nine months as well.

Finally, I'd like to thank my colleagues at YPlan, my friends, my family, and especially my wife, Esther, for supporting me and for pulling me away from the command line at just the right times.

CHAPTER 1

Introduction

This book is about doing data science at the command line. My aim is to make you a more efficient and productive data scientist by teaching you how to leverage the power of the command line.

Having both *data science* and *command line* in the book's title requires an explanation. How can a technology that is more than 50 years old[1] be of any use to a field that is only a few years young?

Today, data scientists can choose from an overwhelming collection of exciting technologies and programming languages. Python, R, Julia, and Apache Spark are but a few examples. You may already have experience in one or more of these. And if so, why should you still care about the command line for doing data science? What does the command line have to offer that these other technologies and programming languages do not?

These are valid questions. In this opening chapter I will answer these questions as follows. First, I provide a practical definition of data science that will act as the backbone of this book. Second, I'll list five important advantages of the command line. By the end of this chapter, I hope to have convinced you that the command line is indeed worth learning for doing data science.

1 The development of the UNIX operating system started back in 1969. It featured a command line since the beginning. The important concept of pipes, which I will discuss in "Essential Unix Concepts" on page 13, was added in 1973.

Data Science Is OSEMN

The field of data science is still in its infancy, and as such, there exist various definitions of what it encompasses. Throughout this book I employ a very practical definition devised by Hilary Mason and Chris H. Wiggins.[2] They define data science according to the following five steps: (1) obtaining data, (2) scrubbing data, (3) exploring data, (4) modeling data, and (5) interpreting data. Together, these steps form the OSEMN (pronounced *awesome*) model. This definition serves as the backbone of this book because each step (except for step 5, interpreting data, which I'll explain shortly) has its own chapter.

Although the five steps are discussed in a linear and incremental fashion, in practice it is very common to move back and forth between them or to perform multiple steps at the same time. Figure 1-1 illustrates that doing data science is an iterative and nonlinear process. For example, once you have modeled your data and have looked at the results, you may decide to go back to the scrubbing step to adjust the features of the dataset.

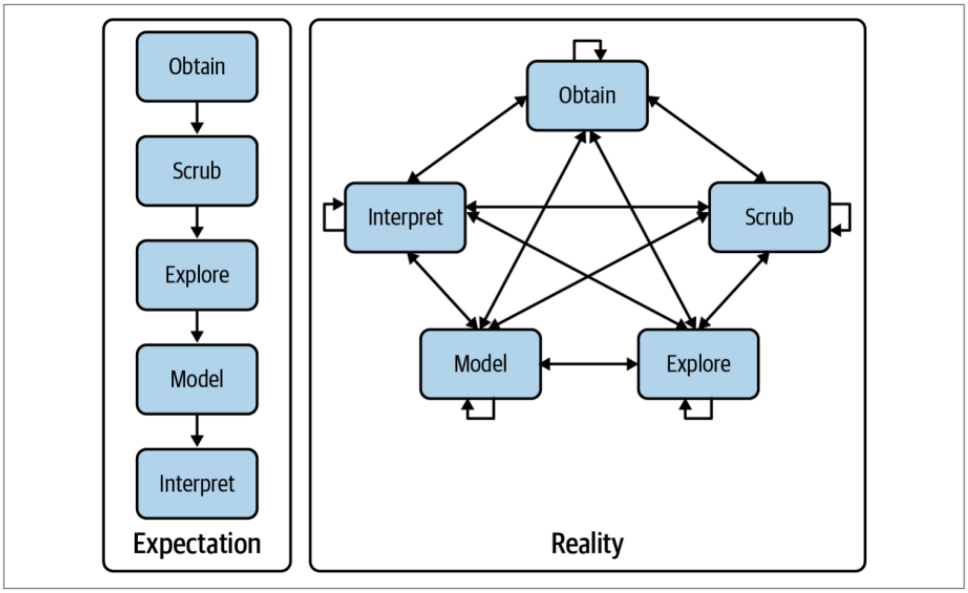


Figure 1-1. Doing data science is an iterative and nonlinear process

In the following pages, I explain what each step entails.

2 "A Taxonomy of Data Science," *dataists* (blog), September 25, 2010, *http://www.dataists.com/2010/09/a-taxonomy-of-data-science*.

Obtaining Data

Without any data, there is little data science you can do. So the first step is obtaining data. Unless you are fortunate enough to already possess data, you may need to do one or more of the following:

- Download data from another location (e.g., a web page or server)
- Query data from a database or API (e.g., MySQL or Twitter)
- Extract data from another file (e.g., an HTML file or spreadsheet)
- Generate data yourself (e.g., reading sensors or taking surveys)

In Chapter 3, I discuss several methods for obtaining data using the command line. The obtained data will most likely be in plain text, CSV, JSON, HTML, or XML format. The next step is to scrub this data.

Scrubbing Data

It is not uncommon for the obtained data to have missing values, inconsistencies, errors, weird characters, or uninteresting columns. In such cases, you have to *scrub*, or clean, the data before you can do anything interesting with it. Common scrubbing operations include:

- Filtering lines
- Extracting certain columns
- Replacing values
- Extracting words
- Handling missing values and duplicates
- Converting data from one format to another

While we data scientists love to create exciting data visualizations and insightful models (steps 3 and 4 of the OSEMN model), usually much effort goes into obtaining and scrubbing the required data first (steps 1 and 2). In *Data Jujitsu*(O'Reilly), DJ Patil states that "80% of the work in any data project is in cleaning the data." In Chapter 5, I demonstrate how the command line can help accomplish such data scrubbing operations.

Exploring Data

Once you have scrubbed your data, you are ready to explore it. This is where it gets interesting, because it's when you're exploring that you truly get to know your data. In Chapter 7 I show you how the command line can be used to:

- Look at your data
- Derive statistics from your data
- Create insightful visualizations

Command-line tools used in Chapter 7 include `csvstat` and `rush`.

Modeling Data

If you want to explain your data or predict what will happen, you probably want to create a statistical model of the data. Techniques to create a model include clustering, classification, regression, and dimensionality reduction. The command line is not suitable for programming a new type of model from scratch. It is, however, very useful to be able to build a model from the command line. In Chapter 9 I will introduce several command-line tools that either build a model locally or employ an API to perform the computation in the cloud.

Interpreting Data

The final and perhaps most important step in the OSEMN model is interpreting data. This step involves:

- Drawing conclusions from your data
- Evaluating what your results mean
- Communicating your results

To be honest, the computer is of little use here, and the command line does not really come into play at this stage. Once you have reached this step, it's up to you. This is the only step in the OSEMN model that does not have its own chapter. Instead, I refer you to the book *Thinking with Data* by Max Shron (O'Reilly).

Intermezzo Chapters

Besides the chapters that cover the OSEMN steps, there are four intermezzo chapters. Each discusses a more general topic concerning data science and how the command line is employed for that. These topics are applicable to any step in the data science process.

In Chapter 4, I discuss how to create reusable tools for the command line. These personal tools can come from long commands that you have typed on the command line or from existing code that you have written in, say, Python or R. Being able to create your own tools allows you to become more efficient and productive.

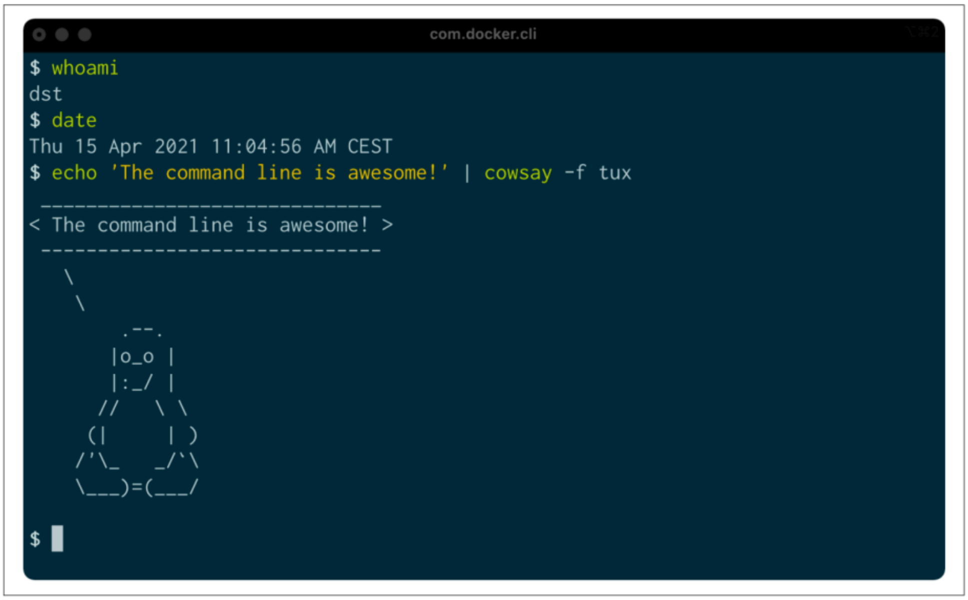
Because the command line is an interactive environment for doing data science, it can become challenging to keep track of your workflow. In Chapter 6, I demonstrate a command-line tool called `make`, which allows you to define your data science workflow in terms of tasks and the dependencies between them. This tool increases the reproducibility of your workflow, not only for you but also for your colleagues and peers.

In Chapter 8, I explain how your commands and tools can be sped up by running them in parallel. Using a command-line tool called GNU Parallel, you can apply command-line tools to very large datasets and run them on multiple cores or even on remote machines.

In Chapter 10, I discuss how to employ the power of the command line in other environments and programming languages, such as R, RStudio, Python, Jupyter Notebooks, and even Apache Spark.

What Is the Command Line?

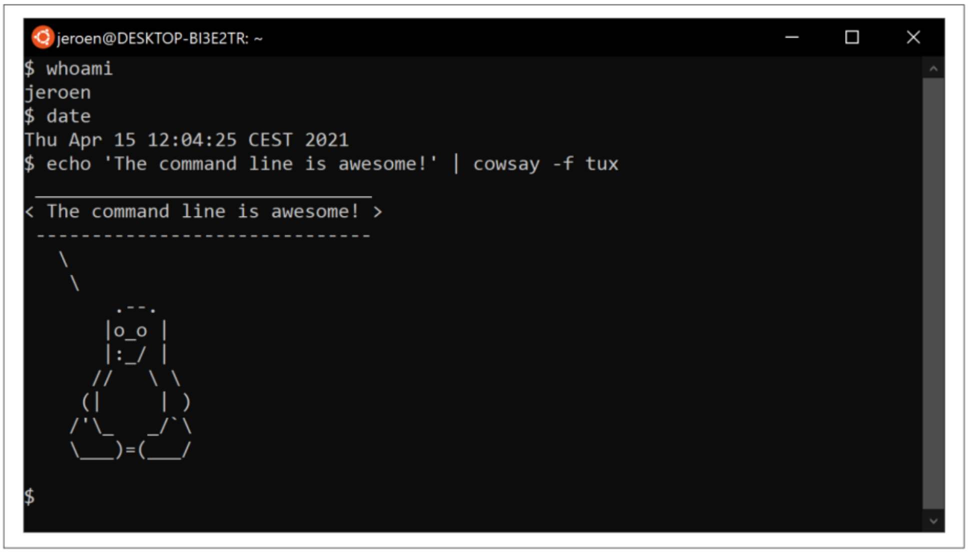
Before I discuss *why* you should use the command line for data science, let's take a peek at *what* the command line actually looks like (it may be already familiar to you). Figures 1-2 and 1-3 show a screenshot of the command line as it appears by default on macOS and Ubuntu, respectively. Ubuntu is a particular distribution of GNU/Linux, and it's the one I'll be using in this book.



```
$ whoami
dst
$ date
Thu 15 Apr 2021 11:04:56 AM CEST
$ echo 'The command line is awesome!' | cowsay -f tux
 ______________________________
< The command line is awesome! >
 ------------------------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/

$
```

Figure 1-2. Command line on macOS



```
$ whoami
jeroen
$ date
Thu Apr 15 12:04:25 CEST 2021
$ echo 'The command line is awesome!' | cowsay -f tux
 ______________________________
< The command line is awesome! >
 ------------------------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/

$
```

Figure 1-3. Command line on Ubuntu

The window shown in the two screenshots is called the *terminal*. This is the program that enables you to interact with the *shell*. It is the shell that executes the commands you type in. In Chapter 2, I explain these two terms in more detail.

> I'm not showing the Microsoft Windows command line (also known as the Command Prompt or PowerShell), because it's fundamentally different from and incompatible with the commands presented in this book. The good news is that you can install a Docker image on Microsoft Windows so that you're able to follow along. Installation of the Docker image is explained in Chapter 2.

Interacting with your computer by typing commands is very different from going through a *graphical user interface* (GUI). If you are mostly used to processing data in, say, Microsoft Excel, then this approach may seem intimidating at first. Don't be afraid. Trust me when I say that you'll get used to working at the command line very quickly.

In this book, the commands that I type and the output that they generate are displayed as text. For example, the contents of the terminal in the two screenshots would look like this:

```
$ whoami
dst

$ date
Tue Jun 29 02:25:17 PM CEST 2021
```

```
$ echo 'The command line is awesome!' | cowsay -f tux
 ______________________________
< The command line is awesome! >
 ------------------------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/


$
```

You'll notice that each command is preceded by a dollar sign (**$**). This is called the *prompt*. The prompt in the two screenshots shows more information, namely the username, the date, and a penguin. It's a convention to show only a dollar sign in examples, because the prompt (1) can change during a session (when you go to a different directory), (2) can be customized by the user (e.g., it can also show the time or the current `git`[3] branch you're working on), and (3) is irrelevant for the commands themselves.

In the next chapter I'll explain much more about essential command-line concepts. But first, it's time to explain *why* you should learn to use the command line for doing data science.

Why Data Science at the Command Line?

The command line has many great advantages that can really make you a more efficient and productive data scientist. Roughly grouping the advantages, the command line is *agile*, *augmenting*, *scalable*, *extensible*, and *ubiquitous*.

The Command Line Is Agile

The first advantage of the command line is that it allows you to be agile. Data science has a very interactive and exploratory nature, and the environment that you work in needs to allow for that. The command line achieves this by two means.

First, the command line provides a so-called *read-eval-print loop* (REPL). This means that you type in a command, press Enter, and the command is evaluated immediately.

3 Linus Torvalds and Junio C. Hamano, *git – the Stupid Content Tracker*, version 2.25.1, 2021, *https://git-scm.com*.

A REPL is often much more convenient for doing data science than the edit-compile-run-debug cycle associated with scripts, large programs, and, say, Hadoop jobs. Your commands are executed immediately, may be stopped at will, and can be changed quickly. This short iteration cycle really allows you to play with your data.

Second, the command line is very close to the filesystem. Because data is the main ingredient for doing data science, it is important to be able to work easily with the files that contain your dataset. The command line offers many convenient tools for this.

The Command Line Is Augmenting

The command line integrates well with other technologies. Whatever technology your data science workflow currently includes (whether it's R, Python, or Excel), please know that I'm not suggesting you abandon that workflow. Instead, consider the command line as an augmenting technology that amplifies the technologies you're currently employing. It can do so in three ways.

First, the command line can act as a glue between many different data science tools. One way to glue tools is by connecting the output from the first tool to the input of the second tool. In Chapter 2 I explain how this works.

Second, you can often delegate tasks to the command line from your own environment. For example, Python, R, and Apache Spark allow you to run command-line tools and capture their output. I demonstrate this with examples in Chapter 10.

Third, you can convert your code (e.g., a Python or R script) into a reusable command-line tool. That way, the language that it's written in doesn't matter anymore; it can be used from the command line directly or from any environment that integrates with the command line, as mentioned in the previous paragraph. I explain how to do this in Chapter 4.

In the end, every technology has its strengths and weaknesses, so it's good to know several technologies and use the one that is most appropriate for the task at hand. Sometimes that means using R, sometimes the command line, and sometimes even pen and paper. By the end of this book you'll have a solid understanding of when you should use the command line, and when you're better off continuing with your favorite programming language or statistical computing environment.

The Command Line Is Scalable

As I've said before, working on the command line is very different from using a GUI. On the command line you do things by typing, whereas with a GUI you do things by pointing and clicking with a mouse.

Everything that you type manually on the command line can also be automated through scripts and tools. This makes it very easy to rerun your commands if you made a mistake, when the input data has changed, or because your colleague wants to perform the same analysis. Moreover, your commands can be run at specific intervals, on a remote server, and in parallel on many chunks of data (more on that in Chapter 8).

Because the command line is automatable, it becomes scalable and repeatable. It's not straightforward to automate pointing and clicking, which makes a GUI a less suitable environment for doing scalable and repeatable data science.

The Command Line Is Extensible

The command line itself was invented over 50 years ago. Its core functionality has largely remained unchanged, but its *tools*, which are the workhorses of the command line, are being developed on a daily basis.

The command line itself is language agnostic. This allows the command-line tools to be written in many different programming languages. The open source community is producing many free and high-quality command-line tools that we can use for data science.

These command-line tools can work together, which makes the command line very flexible. You can also create your own tools, allowing you to extend the effective functionality of the command line.

The Command Line Is Ubiquitous

Because the command line comes with any Unix-like operating system, including Ubuntu Linux and macOS, it can be found in many places. Plus, 100% of the top five hundred supercomputers are running Linux.[4] So if you ever get your hands on one of those supercomputers (or if you ever find yourself in Jurassic Park with the door locks not working), you'd better know your way around the command line!

But Linux doesn't run only on supercomputers. It also runs on servers, laptops, and embedded systems. These days, many companies offer cloud computing, where you can easily launch new machines on the fly. If you ever log in to such a machine (or a server in general), it's almost certain that you'll arrive at the command line.

It's also important to note that the command line isn't just hype. This technology has been around for more than five decades, and I'm convinced that it's here to stay for another five. Learning how to use the command line (for data science and in general) is therefore a worthwhile investment.

4 See TOP500, which keeps track of how many supercomputers run Linux.

Summary

In this chapter I have introduced you to the OSEMN model for doing data science, which I use as a guide throughout the book. I have provided some background about the Unix command line and hopefully convinced you that it's a suitable environment for doing data science. In the next chapter I'll show you how to get started by installing the datasets and tools and explain the fundamental concepts.

For Further Exploration

- The book *UNIX: A History and a Memoir* by Brian W. Kernighan (self-published) tells the story of Unix, explaining what it is, how it was developed, and why it matters.
- In 2018, I gave a presentation titled "50 Reasons to Learn the Shell for Doing Data Science" at Strata London. You can read the slides if you need even more convincing.
- The short but sweet book *Thinking with Data* by Max Shron (O'Reilly) focuses on the *why* instead of the *how* and provides a framework for defining your data science project that will help you ask the right questions and solve the right problems.

CHAPTER 2

Getting Started

In this chapter, I'm going to make sure that you have all the prerequisites for doing data science at the command line. The prerequisites are threefold: (1) having the same datasets that I use in this book, (2) having a proper environment with all the command-line tools that I use throughout this book, and (3) understanding the essential concepts that come into play when using the command line.

First, I describe how to download the datasets. Second, I explain how to install the Docker image, which is a virtual environment based on Ubuntu Linux that contains all the necessary command-line tools. Finally, I go over the essential Unix concepts through examples.

By the end of this chapter, you'll have everything you need to continue with the first step of doing data science, namely obtaining data.

Getting the Data

The datasets I use in this book can be obtained as follows:

1. Download the ZIP file from the book's website.
2. Create a new directory. You can give this directory any name you like, but I recommend you stick to lowercase letters, numbers, and maybe a hyphen or an underscore so that the name is easier to work with at the command line—for example, *dsatcl2*. Remember where this directory is.
3. Move the ZIP file to that new directory and unpack it.
4. This directory now contains one subdirectory per chapter.

In the next section I explain how to install the environment containing all the command-line tools to work with this data.

Installing the Docker Image

In this book we use many different command-line tools. Unix often comes with a lot of command-line tools preinstalled and offers many packages that contain more relevant tools. Installing these packages yourself is often not too difficult. However, we'll also use tools that are not available as packages and require a more manual and more involved installation. So that you can acquire the necessary command-line tools without having to go through the installation process for each tool, I encourage you, whether you're on Windows, macOS, or Linux, to install the Docker image that was created specifically for this book.

A Docker image is a bundle of one or more applications together with all their dependencies. A Docker container is an isolated environment that runs an image. You can manage Docker images and containers using the `docker` command-line tool (which is what you'll do below) or the Docker GUI. In a way, a Docker container is like a virtual machine, only a Docker container uses far fewer resources. At the end of this chapter I suggest some resources for learning more about Docker.

> If you still prefer to run the command-line tools natively rather than inside a Docker container, then you can, of course, install the command-line tools individually yourself. The code to build the Docker image can be found on GitHub and may serve as a guide to help you with that. Please be aware that this can be time consuming for some tools, as they require many nontrivial steps, such as compiling from source.

To install the Docker image, you first need to download Docker itself from the Docker website. Once it is installed, you invoke the following command on your terminal or command prompt to download the Docker image (don't type the dollar sign):

```
$ docker pull datasciencetoolbox/dsatcl2e
```

You can run the Docker image as follows:

```
$ docker run --rm -it datasciencetoolbox/dsatcl2e
```

You're now inside an isolated environment known as a *Docker container* that has all the necessary command-line tools installed. If the following command produces an enthusiastic cow, then you know everything is working correctly:

```
$ cowsay "Let's moove\!"
 ______________
< Let's moove! >
 --------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

If you want to get data in and out of the container, you can add a volume, which means that a local directory gets mapped to a directory inside the container. I recommend that you first create a new directory, navigate to this new directory, and then run the following when you're on macOS or Linux:

```
$ docker run --rm -it -v "$(pwd)":/data datasciencetoolbox/dsatcl2e
```

Or run the following when you're on Windows and using the Command Prompt (also known as `cmd`):

```
C:\> docker run --rm -it -v "%cd%":/data datasciencetoolbox/dsatcl2e
```

Or the following when you're using Windows PowerShell:

```
PS C:\> docker run --rm -it -v ${PWD}:/data datasciencetoolbox/dsatcl2e
```

In the above commands, the option `-v` instructs `docker` to map the current directory to the */data* directory inside the container, so this is the place to get data in and out of the Docker container.

If you would like to know more about the Docker image, you can visit it on Docker Hub.

When you're done, you can shut down the Docker container by typing `exit`.

Essential Unix Concepts

In Chapter 1, I briefly showed you what the command line is. Now that you are running the Docker image, we can really get started. In this section, I discuss several concepts and tools that you will need to know to feel comfortable doing data science at the command line. If up until now you have been mainly working with graphical user interfaces, then this might be quite a change. But don't worry—I'll start at the beginning and very gradually go on to more advanced topics.

This section is not a complete course in Unix. I will explain only the concepts and tools that are relevant to doing data science. One of the advantages of the Docker image is that a lot is already set up. If you wish to know more, consult "For Further Exploration" on page 33.

The Environment

So you've just logged in to a brand-new environment. Before you do anything, it's worthwhile to get a high-level understanding of this environment, which is roughly defined by four layers, listed here from the top down:

Command-line tools
: First and foremost, there are the command-line tools that you work with. We use them by typing their corresponding commands. There are different types of command-line tools, which I will discuss in the next section. Examples of tools are `ls`,[1] `cat`,[2] and `jq`.[3]

Terminal
: The terminal, which is the second layer, is the application that we type our commands in. If you see the following text mentioned in the book:

```
$ seq 3
1
2
3
```

then you would type **`seq 3`** into your terminal and press Enter. (The command-line tool `seq`,[4] as you can see, generates a sequence of numbers.) You do not type the dollar sign (`$`). It's just there to tell you that this is a command you can type in the terminal. This dollar sign is known as the *prompt*. The text below `seq 3` is the output of the command.

Shell
: The third layer is the shell. Once we have typed in our command and pressed Enter, the terminal sends that command to the shell. The shell is a program that interprets the command. I use the Z shell, but many other shells are available, such as Bash and Fish.

1 Richard M. Stallman and David MacKenzie, *ls – List Directory Contents*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

2 Torbjorn Granlund and Richard M. Stallman, *cat – Concatenate Files and Print on the Standard Output*, version 8.30, 2018, *https://www.gnu.org/software/coreutils*.

3 Stephen Dolan, *jq – Command-Line JSON Processor*, version 1.6, 2021, *https://stedolan.github.io/jq/*

4 Ulrich Drepper, *seq – Print a Sequence of Numbers*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

Operating system
: The fourth layer is the operating system, which is GNU/Linux in our case. Linux is the name of the kernel, which is the heart of the operating system. The kernel is in direct contact with the CPU, disks, and other hardware. The kernel also executes our command-line tools. GNU, which stands for "GNU's not UNIX," refers to the set of basic tools. The Docker image is based on a particular GNU/Linux distribution called Ubuntu.

Executing a Command-Line Tool

Now that you have a basic understanding of the environment, it is high time that you try out some commands. Type the following in your terminal (without the dollar sign) and press Enter:

```
$ pwd
/home/dst
```

You just executed a command that contained a single command-line tool. The tool `pwd`[5] outputs the name of the directory where you currently are. By default, when you log in, this is your home directory.

The command-line tool `cd`, which is a Z shell builtin, allows you to navigate to a different directory:

```
$ cd /data/ch02 ❶

$ pwd ❷
/data/ch02

$ cd .. ❸

$ pwd ❹
/data

$ cd ch02 ❺
```

❶ Navigate to the directory */data/ch02*.

❷ Print the current directory.

❸ Navigate to the parent directory.

❹ Print the current directory again.

5 Jim Meyering, *pwd – Print Name of Current/Working Directory*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

❺ Navigate to the subdirectory *ch02*.

The part after `cd` specifies the directory you want to navigate to. Values that come after the command are called *command-line arguments* or *options*. The two dots refer to the parent directory. One dot, by the way, refers to the current directory. While `cd .` wouldn't have any effect, you'll still see one dot being used in other places.

Let's try a different command:

```
$ head -n 3 movies.txt
Matrix
Star Wars
Home Alone
```

Here we pass three command-line arguments to `head`.[6] The first one is an option. Here I used the short option `-n`. Sometimes a short option has a long variant, which would be `--lines` in this case. The second one is a value that belongs to the option. The third one is a filename. This particular command outputs the first three lines of the file */data/ch02/movies.txt*.

Five Types of Command-Line Tools

I use the term *command-line tool* a lot, but I haven't yet explained what I actually mean by it. I use command-line tool as an umbrella term for *anything* that can be executed from the command line (see Figure 2-1). Under the hood, each command-line tool is one of the following five types:

- A binary executable
- A shell builtin
- An interpreted script
- A shell function
- An alias

6 David MacKenzie and Jim Meyering, *head – Output the First Part of Files*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

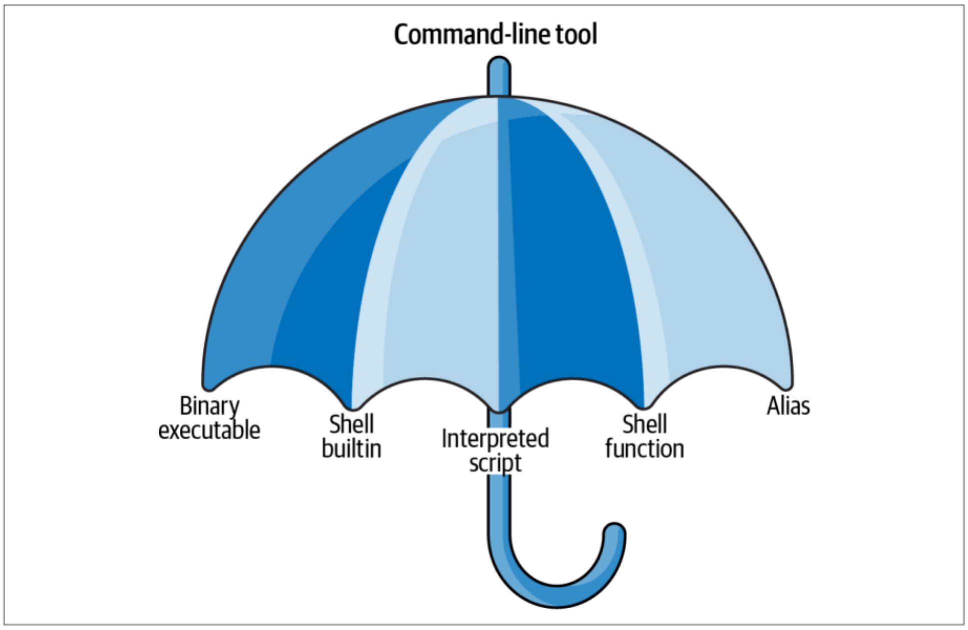


Figure 2-1. I use the term "command-line tool" as an umbrella term

It's good to know the difference between the types. The command-line tools that come preinstalled with the Docker image mostly comprise the first two types (binary executable and shell builtin). The other three types (interpreted script, shell function, and alias) allow us to further build up our data science toolbox and become more efficient and more productive data scientists:

Binary executable
: Binary executables are programs in the classical sense. A binary executable is created by compiling source code to machine code. This means that when you open the file in a text editor, you cannot read it.

Shell builtin
: Shell builtins are command-line tools provided by the shell, which is the Z shell (or `zsh`) in our case. Examples include `cd` and `pwd`. Shell builtins may differ between shells. Like binary executables, they cannot be easily inspected or changed.

Interpreted script
: An interpreted script is a text file that is executed by a binary executable. Examples include Python, R, and Bash scripts. One great advantage of an interpreted script is that you can read and change it. The following script is interpreted by Python not because of the file extension *.py* but because the first line of the script defines the binary that should execute it:

```
$ bat fac.py
───────┬────────────────────────────────────────────────────────
       │ File: fac.py
───────┼────────────────────────────────────────────────────────
       │ #!/usr/bin/env python
       │
       │ def factorial(x):
       │     result = 1
       │     for i in range(2, x + 1):
       │         result *= i
       │     return result
       │
       │ if __name__ == "__main__":
       │     import sys
       │     x = int(sys.argv[1])
       │     sys.stdout.write(f"{factorial(x)}\n")
───────┴────────────────────────────────────────────────────────
```

This script computes the factorial of the integer that we pass as a parameter. It can be invoked from the command line as follows:

```
$ ./fac.py 5
120
```

In Chapter 4, I'll discuss in great detail how to create reusable command-line tools using interpreted scripts.

Shell function

A shell function is a function that is executed by the shell itself (zsh, in our case). Shell functions provide similar functionality to a script, but they are usually (but not necessarily) smaller than scripts. They also tend to be more personal. The following command defines a function called `fac`, which, just like the interpreted Python script I just described, computes the factorial of the integer we pass as a parameter. It does so by generating a list of numbers using `seq`, putting those numbers on one line with `*` as the delimiter using `paste`,[7] and passing this equation into `bc`,[8] which evaluates it and outputs the result:

```
$ fac() { (echo 1; seq $1) | paste -s -d\* - | bc; }

$ fac 5
120
```

7 David M. Ihnat and David MacKenzie, *paste – Merge Lines of Files*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

8 Philip A. Nelson, *bc – an Arbitrary Precision Calculator Language*, version 1.07.1, 2017, *https://www.gnu.org/software/bc*.

The file *~/.zshrc*, which is a configuration file for the Z shell, is a good place to define your shell functions so that they are always available.

Alias

Aliases are like macros. If you often find yourself executing a certain command with some or all of the same parameters, you can define an alias for the command to save time. An alias is also very useful when you continue to misspell a certain command (Chris Wiggins maintains a useful list of aliases). The following command defines such an alias:

```
$ alias l='ls --color -lhF --group-directories-first'

$ alias les=less
```

Now, if you type the following on the command line, the shell will replace each alias it finds with its value:

```
$ cd /data

$ l
total 40K
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch01/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch02/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch03/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch04/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch05/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch06/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch07/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch08/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch09/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch10/

$ cd ch02
```

Aliases are simpler than shell functions, as they don't allow parameters. The function `fac` could not have been defined using an alias because of the parameter. Still, aliases allow you to save lots of keystrokes. Like shell functions, aliases are often defined in the file *.zshrc*, which is located in your home directory. To see all aliases currently defined, you run `alias` without arguments. Try it. What do you see?

In this book I focus mostly on the last three types of command-line tools: interpreted scripts, shell functions, and aliases. I do so because these tools can easily be changed. The purpose of a command-line tool is to make your life easier and to make you a more productive and more efficient data scientist. You can find out the type of a command-line tool with `type` (which is itself a shell builtin):

```
$ type -a pwd
pwd is a shell builtin
pwd is /usr/bin/pwd
pwd is /bin/pwd

$ type -a cd
cd is a shell builtin

$ type -a fac
fac is a shell function

$ type -a l
l is an alias for ls --color -lhF --group-directories-first
```

`type` returns three command-line tools for `pwd`. In that case, the first reported command-line tool is used when you type **`pwd`**. In the next section we'll look at how to combine command-line tools.

Combining Command-Line Tools

Because most command-line tools adhere to the Unix philosophy,[9] they are designed to do only one thing, and to do it really well. For example, the command-line tool `grep`[10] can filter lines, `wc`[11] can count lines, and `sort`[12] can sort lines. The power of the command line comes from its ability to combine these small yet powerful command-line tools.

This power is made possible by managing the communication streams of these tools. Each tool has three standard communication streams: standard input, standard output, and standard error. These are often abbreviated as `stdin`, `stdout`, and `stderr`.

Both the standard output and standard error are, by default, redirected to the terminal, so that both normal output and any error messages are printed on the screen. Figure 2-2 illustrates this for both `pwd` and `rev`.[13] If you run `rev`, you'll see that nothing happens. That's because `rev` expects input, which by default is any keys pressed on the keyboard. Try typing a sentence and pressing Enter—`rev` immediately responds with your input in reverse. You can stop sending input by pressing Ctrl-D after which `rev` will stop.

9 Eric S. Raymond, *The Art of Unix Programming* (Addison-Wesley).

10 Jim Meyering, *grep – Print Lines That Match Patterns*, version 3.4, 2019, *https://www.gnu.org/software/grep*.

11 Paul Rubin and David MacKenzie, *wc – Print Newline, Word, and Byte Counts for Each File*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

12 Mike Haertel and Paul Eggert, *sort – Sort Lines of Text Files*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

13 Karel Zak, *rev – Reverse Lines Characterwise*, version 2.36.1, 2021, *https://www.kernel.org/pub/linux/utils/util-linux*.

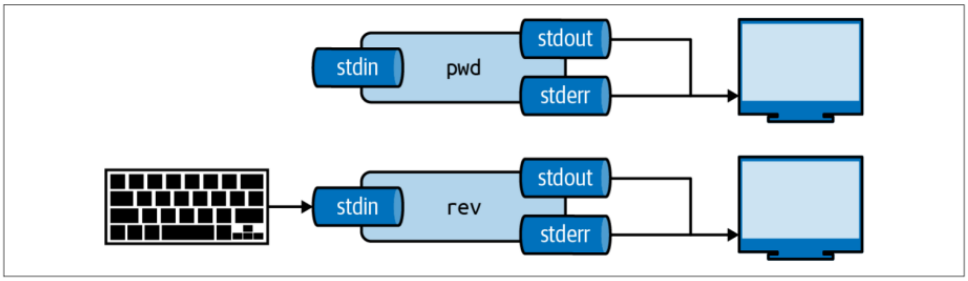


Figure 2-2. Every tool has three standard streams: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`)

In practice, rather than using the keyboard as a source of input, you'll use the output generated by other tools and the contents of files. For example, with `curl` we can download the book *Alice's Adventures in Wonderland* by Lewis Carroll and *pipe* that to the next tool; Figure 2-3 illustrates piping the output from one tool to another tool. (I'll discuss `curl` in more detail in Chapter 3.) This is done using the pipe operator (`|`).

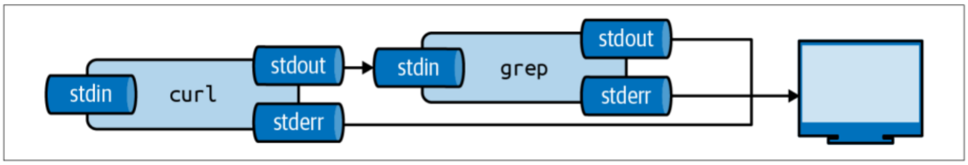


Figure 2-3. The output from a tool can be piped to another tool

We can pipe the output of `curl` to `grep` to filter lines on a pattern. Imagine that we want to see the chapters listed in the table of contents—we can combine `curl` and `grep` as follows:

```
$ curl -s "https://www.gutenberg.org/files/11/11-0.txt" | grep " CHAPTER"
 CHAPTER I.     Down the Rabbit-Hole
 CHAPTER II.    The Pool of Tears
 CHAPTER III.   A Caucus-Race and a Long Tale
 CHAPTER IV.    The Rabbit Sends in a Little Bill
 CHAPTER V.     Advice from a Caterpillar
 CHAPTER VI.    Pig and Pepper
 CHAPTER VII.   A Mad Tea-Party
 CHAPTER VIII.  The Queen's Croquet-Ground
 CHAPTER IX.    The Mock Turtle's Story
 CHAPTER X.     The Lobster Quadrille
 CHAPTER XI.    Who Stole the Tarts?
 CHAPTER XII.   Alice's Evidence
```

And if we want to know *how many* chapters the book has, we can use `wc`, which is very good at counting things:

```
$ curl -s "https://www.gutenberg.org/files/11/11-0.txt" |
> grep " CHAPTER" |
> wc -l ❶
12
```

❶ The option `-l` specifies that `wc` should output only the number of lines that are passed into it. By default, it also returns the number of characters and words.

You can think of piping as an automated copy and paste. Once you get the hang of combining tools using the pipe operator, you'll find that there are virtually no limits to the combinations you can make.

Redirecting Input and Output

Besides piping the output from one tool to another tool, you can also save it to a file. The file will be saved in the current directory, unless a full path is given. This is called *output redirection*, and it works as follows:

```
$ curl "https://www.gutenberg.org/files/11/11-0.txt" | grep " CHAPTER" > chapter
s.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  170k  100  170k    0     0   231k      0 --:--:-- --:--:-- --:--:--  231k

$ cat chapters.txt
 CHAPTER I.     Down the Rabbit-Hole
 CHAPTER II.    The Pool of Tears
 CHAPTER III.   A Caucus-Race and a Long Tale
 CHAPTER IV.    The Rabbit Sends in a Little Bill
 CHAPTER V.     Advice from a Caterpillar
 CHAPTER VI.    Pig and Pepper
 CHAPTER VII.   A Mad Tea-Party
 CHAPTER VIII.  The Queen's Croquet-Ground
 CHAPTER IX.    The Mock Turtle's Story
 CHAPTER X.     The Lobster Quadrille
 CHAPTER XI.    Who Stole the Tarts?
 CHAPTER XII.   Alice's Evidence
```

Here, we save the output of `grep` to a file named *chapters.txt* in the directory */data/ch02*. If this file does not exist yet, it will be created. If this file already exists, its contents are overwritten. Figure 2-4 illustrates how output redirection works conceptually. Note that the standard error is still redirected to the terminal.

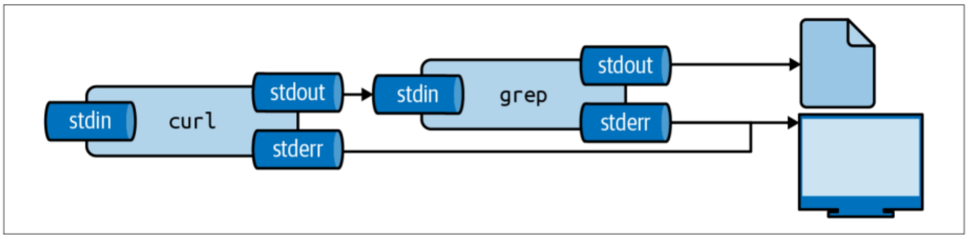


Figure 2-4. The output from a tool can be redirected to a file

You can also append the output to a file with `>>`, meaning the output is added after the original contents:

```
$ echo -n "Hello" > greeting.txt

$ echo " World" >> greeting.txt
```

The tool `echo` outputs the value you specify. The `-n` option, which stands for *newline*, specifies that `echo` should not output a trailing newline.

Saving the output to a file is useful if you need to store intermediate results, for example, to continue with your analysis at a later stage. To use the contents of the file *greeting.txt* again, we can use `cat`, which reads a file and prints it:

```
$ cat greeting.txt
Hello World

$ cat greeting.txt | wc -w ❶
2
```

❶ The `-w` option instructs `wc` to count only words.

The same result can be achieved by using the less-than sign (`<`):

```
$ < greeting.txt wc -w
2
```

This way, you are directly passing the file to the standard input of `wc` without running an additional process.[14] Figure 2-5 illustrates how these two ways work. Again, the final output is the same.

14 Some consider this a "useless use" of `cat`, arguing that the purpose of `cat` is to concatenate files, and not using it for that purpose is a waste of time and costs you a process. I think this is silly. We've got more important things to do!

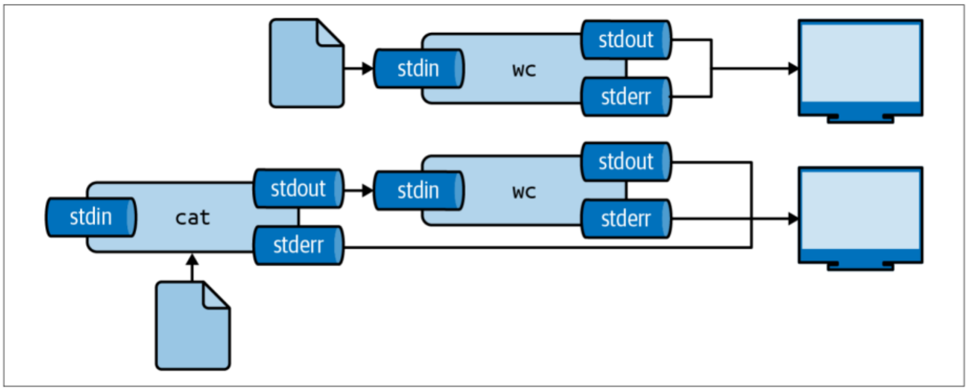


Figure 2-5. Two ways to use the contents of a file as input

Like many command-line tools, `wc` allows one or more filenames to be specified as arguments—for example:

```
$ wc -w greeting.txt movies.txt
 2 greeting.txt
11 movies.txt
13 total
```

Note that in this case, `wc` also outputs the names of the files.

You can suppress the output of any tool by redirecting it to a special file called */dev/null*. I often do this to suppress error messages (see Figure 2-6 for an illustration). The following causes `cat` to produce an error message because it cannot find the file *404.txt*:

```
$ cat movies.txt 404.txt
Matrix
Star Wars
Home Alone
Indiana Jones
Back to the Future
cat: 404.txt: No such file or directory
```

You can redirect standard error to */dev/null* as follows:

```
$ cat movies.txt 404.txt 2> /dev/null  ❶
Matrix
Star Wars
Home Alone
Indiana Jones
Back to the Future
```

❶ The 2 refers to standard error.

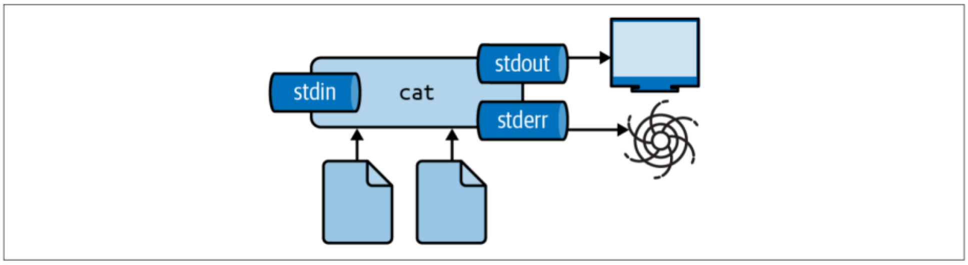


Figure 2-6. Redirecting `stderr` *to* /dev/null

Be careful not to read from and write to the same file. If you do, you'll end up with an empty file. That's because the tool whose output is redirected immediately opens that file for writing, thereby emptying it. There are two work-arounds for this: (1) write to a different file and rename it afterward with `mv`, or (2) use `sponge`,[15] which soaks up all its input before writing to a file. Figure 2-7 illustrates how this works.

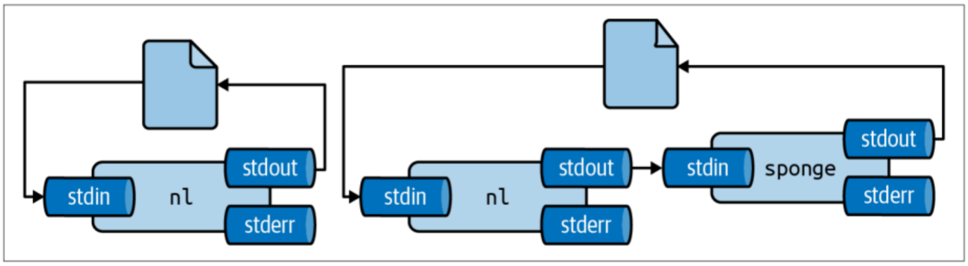


Figure 2-7. Unless you use `sponge`*, you cannot read from and write to the same file in one pipeline*

For example, imagine that you have used `dseq`[16] to generate a file *dates.txt*, and now you'd like to add line numbers using `nl`.[17] If you run the following, the file *dates.txt* will end up empty:

```
$ dseq 5 > dates.txt

$ < dates.txt nl > dates.txt
```

15 Colin Watson and Tollef Fog Heen, *sponge – Soak Up Standard Input and Write to a File*, version 0.65, 2021, *https://joeyh.name/code/moreutils*.

16 Jeroen Janssens, *dseq – Generate Sequence of Dates*, version 0.1, 2021, *https://github.com/jeroenjanssens/dsutils*.

17 Scott Bartram and David MacKenzie, *nl – Number Lines of Files*, version 8.30, 2020, *https://www.gnu.org/software/coreutils*.

```
$ bat dates.txt
───────┬────────────────────────────────────────────────────────────────
       │ File: dates.txt   <EMPTY>
───────┴────────────────────────────────────────────────────────────────
```

Instead, you can use one of the work-arounds I just described:

```
$ dseq 5 > dates.txt

$ < dates.txt nl > dates-nl.txt

$ bat dates-nl.txt
───────┬────────────────────────────────────────────────────────────────
       │ File: dates-nl.txt
───────┼────────────────────────────────────────────────────────────────
   1   │      1  2021-06-30
   2   │      2  2021-07-01
   3   │      3  2021-07-02
   4   │      4  2021-07-03
   5   │      5  2021-07-04
───────┴────────────────────────────────────────────────────────────────

$ dseq 5 > dates.txt

$ < dates.txt nl | sponge dates.txt

$ bat dates.txt
───────┬────────────────────────────────────────────────────────────────
       │ File: dates.txt
───────┼────────────────────────────────────────────────────────────────
   1   │      1  2021-06-30
   2   │      2  2021-07-01
   3   │      3  2021-07-02
   4   │      4  2021-07-03
   5   │      5  2021-07-04
───────┴────────────────────────────────────────────────────────────────
```

Working with Files and Directories

As data scientists, we work with a lot of data. This data is often stored in files. It is important to know how to work with files (and the directories they live in) on the command line. Every action that you can do using a GUI can be done with command-line tools (and you can do much more than that). In this section I introduce the most important tools to list, create, move, copy, rename, and delete files and directories.

Listing the contents of a directory can be done with `ls`. If you don't specify a directory, it lists the contents of the current directory. I prefer `ls` to have a long listing format and to have the directories grouped before files. Instead of typing the corresponding options each time, I use the alias `l`:

```
$ ls /data/ch10
alice.txt  count.py  count.R  Untitled1337.ipynb

$ alias l
l='ls --color -lhF --group-directories-first'

$ l /data/ch10
total 176K
-rw-r--r-- 1 dst dst 164K Jun 29 14:25 alice.txt
-rwxr-xr-x 1 dst dst  408 Jun 29 14:25 count.py*
-rw-r--r-- 1 dst dst  460 Jun 29 14:25 count.R
-rw-r--r-- 1 dst dst 1.7K Jun 29 14:25 Untitled1337.ipynb
```

You have already seen how we can create new files by redirecting the output with either > or >>. If you need to move a file to a different directory, you can use `mv`:[18]

```
$ mv hello.txt /data/ch02
```

You can also rename files with `mv`:

```
$ cd data
$ mv hello.txt bye.txt
```

You can also rename or move entire directories. If you no longer need a file, you can delete (or remove) it with `rm`:[19]

```
$ rm bye.txt
```

If you want to remove an entire directory with all its contents, specify the `-r` option, which stands for "recursive":

```
$ rm -r /data/ch02/old
```

If you want to copy a file, use `cp`.[20] This is useful for creating backups:

```
$ cp server.log server.log.bak
```

You can create directories using `mkdir`:[21]

```
$ cd /data

$ mkdir logs

$ l
total 44K
```

18 Mike Parker, David MacKenzie, and Jim Meyering, *mv – Move (Rename) Files*, version 8.30, 2020, *https://www.gnu.org/software/coreutils*.

19 Paul Rubin et al., *rm – Remove Files or Directories*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

20 Torbjorn Granlund, David MacKenzie, and Jim Meyering, *cp – Copy Files and Directories*, version 8.30, 2018, *https://www.gnu.org/software/coreutils*.

21 David MacKenzie, *mkdir – Make Directories*, version 8.30, 2019, *https://www.gnu.org/software/coreutils*.

```
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch01/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch02/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch03/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch04/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch05/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch06/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch07/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch08/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch09/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 ch10/
drwxr-xr-x 2 dst dst 4.0K Jun 29 14:25 logs/
```

Using the command-line tools to manage your files can be scary at first, because you have no graphical overview of the filesystem to provide immediate feedback. There are a few visual file managers that can help with this, such as GNU Midnight Commander, Ranger, and Vifm. These are not installed in the Docker image, but you can install one of them yourself by running `sudo apt install` followed by either `mc`, `ranger`, or `vifm`.

All of these command-line tools accept the `-v` option, which stands for *verbose*, so that they output what's going on. For example:

```
$ mkdir -v backup
mkdir: created directory 'backup'
```

All tools other than `mkdir` also accept the `-i` option, which stands for *interactive*, and which causes the tools to ask you for confirmation. For example:

```
$ rm -i *
zsh: sure you want to delete all 12 files in /data [yn]? n
```

Managing Output

Sometimes a tool or sequence of tools produces too much output to include in the book. Instead of manually altering such output, I prefer to be transparent by piping it through a helper tool. You don't necessarily have to do this, especially if you're interested in the complete output.

Here are the tools I use to make output manageable.

I often use `trim` to limit the output to a given height and width. By default, output is trimmed to 10 lines and the width of the terminal. Pass a negative number to disable trimming the height and/or the width. For example:

```
$ cat /data/ch07/tips.csv | trim 5 25
bill,tip,sex,smoker,day,...
16.99,1.01,Female,No,Sun...
10.34,1.66,Male,No,Sun,D...
21.01,3.5,Male,No,Sun,Di...
```

```
23.68,3.31,Male,No,Sun,D...
... with 240 more lines
```

Other tools that I use to massage the output are `head`, `tail`, `fold`, `paste`, and `column`. The Appendix contains an example for each of these.

If a file or an output contains a comma-separated value, I often pipe it through `csvlook` to turn it into a nice-looking table. If you run `csvlook`, you'll see the complete table. I have redefined `csvlook` such that the table is shortened by `trim`:

```
$ which csvlook
csvlook () {
        /usr/bin/csvlook "$@" | trim | sed 's/- | -/──┼──/g;s/| -/├──/g;s/- |/──
┤/;s/|/│/g;2s/-/─/g'
}

$ csvlook /data/ch07/tips.csv
│  bill │   tip │ sex    │ smoker │ day │ time   │ size │
├───────┼───────┼────────┼────────┼─────┼────────┼──────┤
│ 16.99 │  1.01 │ Female │  False │ Sun │ Dinner │    2 │
│ 10.34 │  1.66 │ Male   │  False │ Sun │ Dinner │    3 │
│ 21.01 │  3.50 │ Male   │  False │ Sun │ Dinner │    3 │
│ 23.68 │  3.31 │ Male   │  False │ Sun │ Dinner │    2 │
│ 24.59 │  3.61 │ Female │  False │ Sun │ Dinner │    4 │
│ 25.29 │  4.71 │ Male   │  False │ Sun │ Dinner │    4 │
│  8.77 │  2.00 │ Male   │  False │ Sun │ Dinner │    2 │
│ 26.88 │  3.12 │ Male   │  False │ Sun │ Dinner │    4 │
... with 236 more lines
```

I use `bat` to show the contents of a file where line numbers and syntax highlighting matter—for example:

```
$ bat /data/ch04/stream.py
───────┬────────────────────────────────────────────────────────────────
       │ File: /data/ch04/stream.py
───────┼────────────────────────────────────────────────────────────────
   1   │ #!/usr/bin/env python
   2   │ from sys import stdin, stdout
   3   │ while True:
   4   │     line = stdin.readline()
   5   │     if not line:
   6   │         break
   7   │     stdout.write("%d\n" % int(line)**2)
   8   │     stdout.flush()
───────┴────────────────────────────────────────────────────────────────
```

Sometimes I add the `-A` option when I want to explicitly point out the spaces, tabs, and newlines in a file.

Occasionally it's useful to write intermediate output to a file. This allows you to inspect any step in your pipeline once it has completed. You can insert the tool `tee` as often as you like in your pipeline. I often use it to inspect a portion of the final output, while writing the complete output to file (see Figure 2-8). Here, the complete output is written to *even.txt*, and the first five lines are printed using `trim`:

```
$ seq 0 2 100 | tee even.txt | trim 5
0
2
4
6
8
... with 46 more lines
```

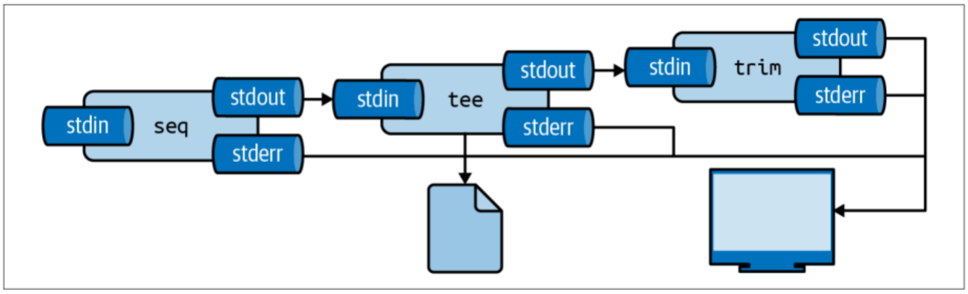


Figure 2-8. With `tee`, you can write intermediate output to a file

Last, to insert images that have been generated by command-line tools (that is, every image other than screenshots and diagrams) I use `display`. If you run `display`, you'll find that it doesn't work. In Chapter 7, I explain four options for displaying images generated from the command line.

Help!

As you're finding your way around the command line, it may happen that you need help. Even the most seasoned users need help at some point. It is impossible to remember all the different command-line tools and their possible arguments. Fortunately, the command line offers severals ways to get help.

Perhaps the most important command for getting help is `man`,[22] which is short for *manual*. It contains information for most command-line tools. In case I've forgotten the options to the tool `tar`, which happens all the time, I just access its manual page using the following:

22 John W. Eaton and Colin Watson, *man – an Interface to the System Reference Manuals*, version 2.9.1, 2020, *https://nongnu.org/man-db*.

```
$ man tar | trim 20
TAR(1)                          GNU TAR Manual                          TAR(1)

NAME
       tar - an archiving utility

SYNOPSIS
   Traditional usage
       tar {A|c|d|r|t|u|x}[GnSkUWOmpsMBiajJzZhPlRvwo] [ARG...]

   UNIX-style usage
       tar -A [OPTIONS] ARCHIVE ARCHIVE

       tar -c [-f ARCHIVE] [OPTIONS] [FILE...]

       tar -d [-f ARCHIVE] [OPTIONS] [FILE...]

       tar -t [-f ARCHIVE] [OPTIONS] [MEMBER...]

       tar -r [-f ARCHIVE] [OPTIONS] [FILE...]

... with 1147 more lines
```

Not every command-line tool has a manual page. Take `cd`, for example:

```
$ man cd
No manual entry for cd
```

For shell builtins like `cd`, you can consult the *zshbuiltins* manual page:

```
$ man zshbuiltins | trim
ZSHBUILTINS(1)             General Commands Manual             ZSHBUILTINS(1)

NAME
       zshbuiltins - zsh built-in commands

SHELL BUILTIN COMMANDS
       Some shell builtin commands take options as described in individual en-
       tries; these are often referred to in the list below as `flags'  to
       avoid  confusion  with  shell options, which may also have an effect on
       the behavior of builtin commands.  In this introductory section,  `op-
... with 2735 more lines
```

You can search by pressing `/` and exit by pressing q. Try to find the appropriate section for `cd`.

Newer command-line tools often lack a manual page as well. In such cases, your best bet is to invoke the tool with the `--help` (or `-h`) option. For example:

```
$ jq --help | trim
jq - commandline JSON processor [version 1.6]

Usage:  jq [options] <jq filter> [file...]
        jq [options] --args <jq filter> [strings...]
        jq [options] --jsonargs <jq filter> [JSON_TEXTS...]

jq is a tool for processing JSON inputs, applying the given filter to
its JSON text inputs and producing the filter's results as JSON on
standard output.

... with 37 more lines
```

Specifying the `--help` option also works for command-line tools such as `cat`. However, the corresponding manual page often provides more information. If, after trying these three approaches, you are still stuck, then consulting the internet is perfectly acceptable. In the Appendix, there's a list of all the command-line tools used in this book. Besides showing how each command-line tool can be installed, the Appendix also shows how you can get help for each tool.

Manual pages can be quite verbose and difficult to read. The tool `tldr`[23] (which is short for "too long; didn't read") is a collection of community-maintained help pages for command-line tools that aims to be a simpler, more approachable complement to traditional manual pages. Here's an example of the tldr page for `tar`:

```
$ tldr tar | trim 20

  tar

  Archiving utility.
  Often combined with a compression method, such as gzip or bzip2.
  More information: https://www.gnu.org/software/tar.

  - [c]reate an archive and write it to a [f]ile:
    tar cf target.tar file1 file2 file3

  - [c]reate a g[z]ipped archive and write it to a [f]ile:
    tar czf target.tar.gz file1 file2 file3

  - [c]reate a g[z]ipped archive from a directory using relative paths:
    tar czf target.tar.gz --directory=path/to/directory .

  - E[x]tract a (compressed) archive [f]ile into the current directory [v]erbos…
    tar xvf source.tar[.gz|.bz2|.xz]

  - E[x]tract a (compressed) archive [f]ile into the target directory:
... with 12 more lines
```

As you can see, rather than listing the many options alphabetically like `man` often does, `tldr` cuts to the chase by giving you a list of practical examples.

23 Owen Voke, *tldr – Collaborative Cheatsheets for Console Commands*, version 3.3.7, 2021, *https://tldr.sh*.

Summary

In this chapter you learned how to get all the required command-line tools by installing a Docker image. I also went over some essential command-line concepts and how to get help. Now that you have all the necessary ingredients, you're ready for the first step of the OSEMN model for data science: obtaining data.

For Further Exploration

- The subtitle of this book pays homage to the epic *Unix Power Tools*, 3rd ed. by Shelley Powers, Jerry Peek, Tim O'Reilly, and Mike Loukides (O'Reilly), and rightly so. With over 51 chapters and more than a thousand pages, *Unix Power Tools* covers just about everything there is to know about Unix. It weighs more than four pounds, so you might want to consider getting the ebook.
- The website explainshell parses a command or a sequence of commands and provides a short explanation of each part. This site is useful for quickly understanding a new command or option without having to skim through the relevant manual pages.
- Docker is truly a brilliant piece of software. In this chapter I've briefly explained how to download a Docker image and run a Docker container, but it might be worthwhile to learn how to create your own Docker images. The book *Docker: Up & Running*, 2nd ed. by Sean P. Kane and Karl Matthias (O'Reilly) is a good resource as well.

CHAPTER 3

Obtaining Data

This chapter deals with the first step of the OSEMN model: obtaining data. After all, without any data, there is not much data science that we can do. I assume that the data you need to solve your data science problem already exists. Your first task is to get this data onto your computer (and possibly also inside the Docker container) in a form that you can work with.

According to the Unix philosophy, text is a universal interface. Almost every command-line tool takes text as input, produces text as output, or both. This is the main reason why command-line tools can work so well together. However, as we'll see, even just text can come in multiple forms.

Data can be obtained in several ways—for example, by downloading it from a server, querying a database, or connecting to a Web API. Sometimes the data comes in a compressed form or in a binary format such as a Microsoft Excel Spreadsheet. In this chapter, I discuss several tools that help tackle this from the command line, including `curl`,[1] `in2csv`,[2] `sql2csv`,[3] and `tar`.[4]

1 Daniel Stenberg, *curl – Transfer a URL*, version 7.68.0, 2016, *https://curl.haxx.se*.

2 Christopher Groskopf, *in2csv – Convert Common, but Less Awesome, Tabular Data Formats to CSV*, version 1.0.5, 2020, *https://csvkit.rtfd.org*.

3 Christopher Groskopf, *sql2csv – Execute an SQL Query on a Database and Output the Result to a CSV File*, version 1.0.5, 2020, *https://csvkit.rtfd.org*.

4 John Gilmore and Jay Fenlason, *tar – an Archiving Utility*, version 1.30, 2014, *https://www.gnu.org/software/tar*.

Overview

In this chapter, you'll learn how to:

- Copy local files to the Docker image
- Download data from the internet
- Decompress files
- Extract data from spreadsheets
- Query relational databases
- Call web APIs

This chapter starts with the following files:

```
$ cd /data/ch03

$ l
total 924K
-rw-r--r-- 1 dst dst 627K Jun 29 14:26 logs.tar.gz
-rw-r--r-- 1 dst dst 189K Jun 29 14:26 r-datasets.db
-rw-r--r-- 1 dst dst  149 Jun 29 14:26 tmnt-basic.csv
-rw-r--r-- 1 dst dst  148 Jun 29 14:26 tmnt-missing-newline.csv
-rw-r--r-- 1 dst dst  181 Jun 29 14:26 tmnt-with-header.csv
-rw-r--r-- 1 dst dst  91K Jun 29 14:26 top2000.xlsx
```

The instructions for getting these files are in Chapter 2. Any other files are either downloaded or generated using command-line tools.

Copying Local Files to the Docker Container

A common situation is that you already have the necessary files on your own computer. This section explains how you can get those files into the Docker container.

I mentioned in Chapter 2 that the Docker container is an isolated virtual environment. Luckily, there is one exception to that: files can be transferred in and out of the Docker container. The local directory from which you ran `docker run` is mapped to a directory in the Docker container. This directory is called */data*. Note that this is not the home directory, which is */home/dst*.

If you have one or more files on your local computer, and you want to apply some command-line tools to them, all you have to do is copy or move the files to that mapped directory. Let's assume that you have a file called *logs.csv* in your Downloads directory.

If you're running Windows, open the Command Prompt or PowerShell and run the following two commands:

```
> cd %UserProfile%\Downloads
> copy logs.csv MyDataScienceToolbox\
```

If you are running Linux or macOS, open a terminal and execute the following command on your operating system (not inside the Docker container):

```
$ cp ~/Downloads/logs.csv ~/my-data-science-toolbox
```

You can also drag and drop the file into the right directory using a graphical file manager such as Windows Explorer or macOS Finder.

Downloading from the Internet

The internet provides, without a doubt, the largest resource for interesting data. The command-line tool `curl` can be considered the command line's Swiss Army knife when it comes to downloading data from the internet.

Introducing curl

When you browse a URL, which stands for *uniform resource locator*, your browser interprets the data it downloads. For example, the browser renders HTML files, plays video files automatically, and shows PDF files. However, when you use `curl` to access a URL, it downloads the data and, by default, prints it to standard output. `curl` doesn't do any interpretation, but luckily other command-line tools can be used to process the data further.

The easiest invocation of `curl` is to specify a URL as a command-line argument. Let's try downloading an article from Wikipedia:

```
$ curl "https://en.wikipedia.org/wiki/List_of_windmills_in_the_Netherlands" |
> trim ❶
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0<!
DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>List of windmills in the Netherlands - Wikipedia</title>
<script>document.documentElement.className="client-js";RLCONF={"wgBreakFrames":…
"wikitext","wgRelevantPageName":"List_of_windmills_in_the_Netherlands","wgRelev…
"site.styles":"ready","noscript":"ready","user.styles":"ready","ext.globalCssJs…
"ext.growthExperiments.SuggestedEditSession"];</script>
<script>(RLQ=window.RLQ||[]).push(function(){mw.loader.implement("user.options@…
100  244k  100  244k    0     0  1291k      0 --:--:-- --:--:-- --:--:-- 1291k
… with 1723 more lines
```

❶ Remember, `trim` is used only to make the output fit nicely in the book.

As you can see, `curl` downloads the raw HTML returned by Wikipedia's server; no interpretation is being done, and the contents are immediately printed on standard output. Because of the URL, you'd think that this article would list all the windmills in the Netherlands. However, there are apparently so many windmills that each province has its own page. Fascinating.

By default, `curl` outputs a progress meter that shows the download rate and the expected time of completion. This output isn't written to standard output but instead is written to a separate channel known as *standard error*, so that it doesn't interfere when you add another tool to the pipeline. While this information can be useful when downloading very large files, I usually find it distracting, so I specify the `-s` option to *silence* this output:

```
$ curl -s "https://en.wikipedia.org/wiki/List_of_windmills_in_Friesland" |
> pup -n 'table.wikitable tr'  ❶
234
```

❶ I'll discuss `pup`,[5] a handy tool for scraping websites, in more detail in Chapter 5.

And what do you know, there are apparently 234 windmills in the province of Friesland alone!

Saving

You can let `curl` save the output to a file by adding the `-O` option. The filename will be based on the last part of the URL:

```
$ curl -s "https://en.wikipedia.org/wiki/List_of_windmills_in_Friesland" -O

$ l
total 1.4M
-rw-r--r-- 1 dst dst 427K Jun 29 14:27 List_of_windmills_in_Friesland
-rw-r--r-- 1 dst dst 627K Jun 29 14:26 logs.tar.gz
-rw-r--r-- 1 dst dst 189K Jun 29 14:26 r-datasets.db
-rw-r--r-- 1 dst dst  149 Jun 29 14:26 tmnt-basic.csv
-rw-r--r-- 1 dst dst  148 Jun 29 14:26 tmnt-missing-newline.csv
-rw-r--r-- 1 dst dst  181 Jun 29 14:26 tmnt-with-header.csv
-rw-r--r-- 1 dst dst  91K Jun 29 14:26 top2000.xlsx
```

If you don't like that filename, then you can use the `-o` option together with a filename or redirect the output to a file yourself:

```
$ curl -s "https://en.wikipedia.org/wiki/List_of_windmills_in_Friesland" > fries
land.html
```

5 Eric Chiang, *pup – Parsing HTML at the Command Line*, version 0.4.0, 2016, *https://github.com/EricChiang/pup*.

Other Protocols

In total, `curl` supports more than 20 protocols. To download from an FTP server (FTP stands for "File Transfer Protocol"), you use `curl` the same way. Here I download the file *welcome.msg* from *ftp.gnu.org*:

```
$ curl -s "ftp://ftp.gnu.org/welcome.msg" | trim
NOTICE (Updated December 18 2018):

FSF public IP addresses are changing between December 20 and January 7th

If you have hardcoded the IP address of any GNU/FSF servers in those
ranges in any code or configuration files, they will need to be
updated. If you refer to our servers by their DNS name, such as
"gnu.org", then that will continue to work. You should use the DNS name
wherever possible.

... with 68 more lines
```

If the specified URL is a directory, `curl` will list the contents of that directory. When the URL is password protected, you can specify a username and a password with the `-u` option.

Or there's the DICT protocol, which allows you to access various dictionaries and look up definitions. Here's the definition of *windmill* according to the Collaborative International Dictionary of English:

```
$ curl -s "dict://dict.org/d:windmill" | trim
220 dict.dict.org dictd 1.12.1/rf on Linux 4.19.0-10-amd64 <auth.mime> <4623255…
250 ok
150 1 definitions retrieved
151 "Windmill" gcide "The Collaborative International Dictionary of English v.0…
Windmill \Wind"mill`\, n.
   A mill operated by the power of the wind, usually by the
   action of the wind upon oblique vanes or sails which radiate
   from a horizontal shaft. --Chaucer.
   [1913 Webster]
.
... with 2 more lines
```

When you are downloading data from the internet, however, the protocol will most likely be HTTP, so the URL will start with either *http://* or *https://*.

Following Redirects

When you access a shortened URL, such as the one that starts with *http://bit.ly/* or *http://t.co/*, your browser automatically redirects you to the correct location. With `curl`, however, you need to specify the `-L` or `--location` option in order to be redirected. If you don't, you can get something like:

```
$ curl -s "https://bit.ly/2XBxvwK"
<html>
<head><title>Bitly</title></head>
<body><a href="https://youtu.be/dQw4w9WgXcQ">moved here</a></body>
</html>%
```

Sometimes you get nothing back, like when we follow the URL just mentioned:

```
$ curl -s "https://youtu.be/dQw4w9WgXcQ"
```

If you specify the `-I` or `--head` option, `curl` fetches only the HTTP header of the response, which allows you to inspect the status code and other information returned by the server:

```
$ curl -sI "https://youtu.be/dQw4w9WgXcQ" | trim
HTTP/2 303
content-type: application/binary
x-content-type-options: nosniff
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: Mon, 01 Jan 1990 00:00:00 GMT
date: Tue, 29 Jun 2021 12:27:13 GMT
location: https://www.youtube.com/watch?v=dQw4w9WgXcQ&feature=youtu.be
content-length: 0
x-frame-options: SAMEORIGIN
… with 9 more lines
```

The first line shows the protocol followed by the HTTP status code, which is 303 in this case. You can also see the location this URL redirects to. Inspecting the header and getting the status code is a useful debugging tool in case `curl` does not give you the expected result. Other common HTTP status codes include 404 (not found) and 403 (forbidden). Wikipedia has a page that lists all HTTP status codes.

In summary, `curl` is a useful command-line tool for downloading data from the internet. Its three most common options are `-s` to silence the progress meter, `-u` to specify a username and password, and `-L` to automatically follow redirects. See the man page for more information (and to make your head spin):

```
$ man curl | trim 20
curl(1)                          Curl Manual                          curl(1)

NAME
       curl - transfer a URL

SYNOPSIS
       curl [options / URLs]

DESCRIPTION
       curl  is  a tool to transfer data from or to a server, using one of the
       supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS,  IMAP,
       IMAPS,  LDAP,  LDAPS,  MQTT, POP3, POP3S, RTMP, RTMPS, RTSP, SCP, SFTP,
       SMB, SMBS, SMTP, SMTPS, TELNET and TFTP). The command  is  designed  to
       work without user interaction.

       curl offers a busload of useful tricks like proxy support, user authen-
       tication, FTP upload, HTTP post, SSL connections, cookies, file  trans-
       fer  resume,  Metalink,  and more. As you will see below, the number of
       features will make your head spin!

… with 3986 more lines
```

Decompressing Files

If the original dataset is very large or is a collection of many files, it may be a compressed archive. Datasets that contain many repeated values (such as the words in a text file or the keys in a JSON file) are especially well suited for compression.

Common file extensions of compressed archives are *.tar.gz*, *.zip*, and *.rar*. To decompress these, you would use the command-line tools `tar`, `unzip`,[6] and `unrar`,[7] respectively. (There are a few less-common file extensions for which you would need other tools.)

Let's take *tar.gz* (pronounced "gzipped tarball") as an example. To extract an archive named *logs.tar.gz*, you would use the following incantation:

```
$ tar -xzf logs.tar.gz    ❶
```

❶ It's common to combine these three short options, like I did here, but you can also specify them separately as `-x -z -f`. In fact, many command-line tools allow you to combine options that consist of a single character.

Indeed, `tar` is notorious for its many command-line arguments. In this case, the three options `-x`, `-z`, and `-f` specify that `tar` should *extract* files from an archive, use gzip as the decompression algorithm, and use the file *logs.tar.gz*.

However, since we're not yet familiar with this archive, it's a good idea to first examine its contents. This can be done using the `-t` option (instead of the `-x` option):

```
$ tar -tzf logs.tar.gz | trim
E1FOSPSAYDNUZI.2020-09-01-00.0dd00628
E1FOSPSAYDNUZI.2020-09-01-00.b717c457
E1FOSPSAYDNUZI.2020-09-01-01.05f904a4
E1FOSPSAYDNUZI.2020-09-01-02.36588daf
E1FOSPSAYDNUZI.2020-09-01-02.6cea8b1d
```

6 Samuel H. Smith et al., *unzip – List, Test, and Extract Compressed Files in a ZIP Archive*, version 6.0, 2009, *http://www.info-zip.org/pub/infozip*.

7 Ben Asselstine, Christian Scheurer, and Johannes Winkelmann, *unrar – Extract Files from Rar Archives*, version 0.0.1, 2014, *https://web.archive.org/web/20080331080828/http://home.gna.org/unrar*.

```
E1FOSPSAYDNUZI.2020-09-01-02.be4bc86d
E1FOSPSAYDNUZI.2020-09-01-03.16f3fa32
E1FOSPSAYDNUZI.2020-09-01-03.1c0a370f
E1FOSPSAYDNUZI.2020-09-01-03.76df64bf
E1FOSPSAYDNUZI.2020-09-01-04.0a1ade1b
… with 2427 more lines
```

It seems that this archive contains a lot of files, and they are not inside a directory. To keep the current directory clean, it's a good idea to first create a new directory using `mkdir` and extract those files there using the `-C` option:

```
$ mkdir logs

$ tar -xzf logs.tar.gz -C logs
```

Let's verify the number of files and some of their contents:

```
$ ls logs | wc -l
2437

$ cat logs/* | trim
#Version: 1.0
#Fields: date time x-edge-location sc-bytes c-ip cs-method cs(Host) cs-uri-stem…
2020-09-01      00:51:54        SEA19-C1        391     206.55.174.150  GET     …
2020-09-01      00:54:59        CPH50-C2        384     82.211.213.95   GET     …
#Version: 1.0
#Fields: date time x-edge-location sc-bytes c-ip cs-method cs(Host) cs-uri-stem…
2020-09-01      00:04:28        DFW50-C1        391     2a03:2880:11ff:9::face:…
#Version: 1.0
#Fields: date time x-edge-location sc-bytes c-ip cs-method cs(Host) cs-uri-stem…
2020-09-01      01:04:14        ATL56-C4        385     2600:1700:2760:da20:548…
… with 10279 more lines
```

Excellent. Now, I understand that you'd like to scrub and explore these log files, but we'll get to that later, in Chapter 5 and Chapter 7.

In time you'll get used to these options, but I'd like to show you an alternative that you might find convenient. Rather than you having to remember the different command-line tools and their options, there's a handy script called `unpack`[8] that will decompress many different formats. `unpack` looks at the extension of the file that you want to decompress and calls the appropriate command-line tool. Now, in order to decompress this same file, you would run:

```
$ unpack logs.tar.gz
```

8 Patrick Brisbin, *unpack – Extract Common File Formats*, version 0.1, 2013, *https://github.com/jeroenjanssens/dsutils*.